NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A RENDERING SYSTEM INDEPENDENT HIGH LEVEL ARCHITECTURE IMPLEMENTATION FOR NETWORKED VIRTUAL ENVIRONMENTS

by

Robert S. List

September 2002

Thesis Advisor: Rudolph P. Darken
Co-Advisor: Joseph A. Sullivan

This thesis done in cooperation with the MOVES Institute.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 2002	Master's Thesis

4. TITLE AND SUBTITLE A RENDERING SYSTEM INDEPENDENT HIGH LEVEL ARCHITECTURE IMPLEMENTATION FOR NETWORKED VIRTUAL ENVIRONMENTS

5. FUNDING NUMBERS

6. AUTHOR (S) Maj Robert S. List

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the U.S. Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(maximum 200 words)***

The High Level Architecture (HLA) is the Department of Defense standard for networking virtual environments. This thesis implements a modular HLA component that can be used independently from the graphics-rendering engine used by the programmer. The modular design of the HLA component allows programmers of virtual environments to rapidly network their existing standalone virtual environments using the DOD standard networking protocol. The HLA component is being used to build a networked virtual environment compatible with Joint Semi-Automated Forces (JSAF). This networked virtual environment will allow a group of human controlled simulations to interact with JSAF controlled entities over common terrain.

14. SUBJECT TERMS	15. NUMBER OF PAGES
High Level Architecture, HLA, Networked Virtual Environments, Joint Semi-Automated Forces, JSAF	71
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500 **Standard Form 298 (Rev. 2-89)**
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

A RENDERING SYSTEM INDEPENDENT HIGH LEVEL ARCHITECTURE IMPLEMENTATION FOR NETWORKED VIRTUAL ENVIRONMENTS

Robert S. List
Major, United States Marine Corps
B.S., North Carolina State University, 1992

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2002

Author: Robert S. List

Approved by: Rudolph P. Darken
Thesis Advisor

Joseph A. Sullivan
Co-Advisor

Christopher S. Eagle
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The High Level Architecture (HLA) is the Department of Defense standard for networking virtual environments. This thesis implements a modular HLA component that can be used independently from the graphics-rendering engine used by the programmer. The modular design of the HLA component allows programmers of virtual environments to rapidly network their existing standalone virtual environments using the DOD standard networking protocol. The HLA component is being used to build a networked virtual environment compatible with Joint Semi-Automated Forces (JSAF). This networked virtual environment will allow a group of human controlled simulations to interact with JSAF controlled entities over common terrain.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGEMENTS

I would like to acknowledge several people for their help and support that enabled me to complete this thesis. First, I would like to thank my family for their support and understanding. I would also like to thank Malachi Wurpts of Southwest Research Institute for his contributions to this thesis. His programming and module design influence have made this a better thesis. I would also like to thank my advisors Dr. Rudy Darken and CDR Joe Sullivan for their guidance and leadership.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. PROBLEM STATEMENT

The High Level Architecture (HLA) is the Department of Defense standard for networking virtual environments. HLA allows a large amount of flexibility and freedom to programmers. However, this flexibility makes implementing HLA applications a complex undertaking. As yet, no tools exist to aid programmers to rapidly implement a networked virtual environment using the most current version of HLA.

Additionally, several versions of the HLA Run Time Infrastructure (RTI) are available through the Department of Defense and commercial industry. While all these RTIs adhere to the HLA specification, applications written to one RTI are not 100 percent compatible with all other RTIs. This incompatibility can cause extensive engineering costs in large-scale simulations, where individual simulations were developed for different RTIs. An open code base is needed that allows access to the RTI for fast HLA integration.

This thesis will implement an HLA module that can be used to network applications over HLA. An existing application will be able to interface with the HLA module to rapidly bring the application into an HLA networked environment. The HLA module will be built in such a way as to make the rendering system independent of the HLA module provided the rendering system is compatible with C++. Since the HLA module is independent of the rendering system, programmers will not be limited when developing new

applications and existing applications will be compatible with this HLA module.

B. APPROACH

This thesis will demonstrate an HLA compliant application written in C++ using the VEGA API from Multigen-Paradigm. However, it will be possible to use other C++ based rendering engines without changing any of the HLA module code.

The Object Model Template (OMT) chosen for this application is the Real-time Platform Reference Federation Object Model (RPR FOM). This OMT was chosen because of its large user base and with the aim to make this application compatible with Joint Semi-Automated Forces (JSAF). The modular design of this project will allow for easy transition to another FOM.

C. THESIS ORGANIZATION

This thesis is organized in the following chapters:

- Chapter I: Introduction. This chapter states the problem for this thesis and gives an overview of the work.
- Chapter II: History of Networked Virtual Environment Architectures. This chapter gives a history of networked virtual environment architectures
- Chapter III: High Level Architecture. This chapter gives an overview of HLA.
- Chapter IV: Implementation. This chapter goes over the details of the application’s design. This chapter discusses the project’s modular design and the application HLA object model.
- Chapter V: Testing and Results. This chapter discusses the results of the project.

- Chapter VI: Conclusions. This chapter contains a general discussion of the conclusions drawn from this project along with proposed future work in this area.

THIS PAGE INTENTIONALLY LEFT BLANK

II. HISTORY OF NETWORKED VIRTUAL ENVIRONMENT ARCHITECTURES

A. OVERVIEW

The history of the development of the High Level Architecture (HLA) can be traced back to two earlier projects: Simulator Networking (SIMNET) and Distributed Interactive Simulation (DIS). Network environment architectures began with the Defense Advanced Research Project Agency (DARPA) SIMNET project. Later, the DIS project defined a standard network protocol that would allow different simulation projects to interact over a network. HLA was developed by the Defense Modeling and Simulation Office (DMSO) in conjunction with industry to create a more flexible and scalable network architecture as a replacement for DIS.

B. SIMULATOR NETWORKING

The DARPA SIMNET project began in 1983. The following shows the purpose behind developing SIMNET.

> SIMNET was started to demonstrate that networks of low cost simulators could allow team training to be carried out on a virtual battlefield. Previously, simulator training was focused on learning individual skills with standalone simulators.[1]

The SIMNET project developed rapidly during the 1980s. The original application for SIMNET was a tank gunnery

[1] Proctor, Michael D (Ed.). (no date). *Web-based Technical Reference on Simulation Interoperability* (online). Available: <http://www.engr.ucf.edu/people/proctor/Interoperability%20Text/Text%20Outline.htm> (29 Aug. 02), Ch. 8.

trainer. For the trainer, four crew stations were linked together; one station for each crewmen in a tank. The project quickly expanded to include multiple tanks, aircraft, fighting vehicles, and command posts. By 1990, there were approximately 260 different simulators in 11 different sites involved in SIMNET.

The SIMNET project was delivered to the Army in 1990 where the Simulation, Training, Instrumentation Command (STRICOM) changed the name to Distributed Interactive Simulation.

C. DISTRIBUTED INTERACTIVE SIMULATION

For the SIMNET project, all the simulators were of a homogenous type and were all developed by one development team lead by DARPA. The need for an architecture to network heterogeneous simulator types was recognized. The DIS project created a network protocol standard that allowed simulators from different projects to communicate with each other. The DIS project defined how data was to be distributed between simulations to make them interoperable.

Work on developing DIS standards was accomplished at semi-annual Workshops on Standards for the Interoperability of Distributed Simulations. Groups of interested volunteers met at these workshops in order to discuss, develop, and publish DIS standards. The original standards for DIS were approved as IEEE Standard 1278 in 1993. These standards defined the Protocol Data Units (PDU) needed to support entity attributes and movement, weapon firing, detonations, and collision detection.

Distributed Interaction Simulation uses a peer-to-peer architecture. Each simulation in a DIS networked virtual environment is linked to all the other member simulations of the virtual environment on a computer network. Member simulations of a DIS networked virtual environment directly broadcast attribute update and interaction PDUs to all other members.

D. HIGH LEVEL ARCHITECTURE

High Level Architecture was designed to replace DIS with a more flexible and scalable network architecture. HLA was sponsored by DMSO and developed by Science Applications International Corporation, Virtual Technology Corporation, Object Sciences Corporation, and Dynamic Animations Systems.[2] In 1998 HLA was set as the standard network simulation architecture for all new DOD networked virtual environment projects. HLA has been criticized because the protocol was not opened to a standards organization, like DIS was, for review while it was being developed.

HLA was developed to address the limitations of DIS. The number of entities in a DIS system is limited because DIS is built on a peer-to-peer model where entity updates are broadcast to the entire network. As the number of entities increase, the congestion in the network increases until the network becomes saturated. HLA combats this problem by using a Run Time Infrastructure (RTI). Simulations send their updates to the RTI, which keeps track of which other simulations are interested in those

[2] Department of Defense, Defense Modeling and Simulation Office. (no date). *High Level Architecture RTI 1.3-Next Generation Programmer's Guide*, Version 4, inside cover.

updates and the RTI then sends the updates to interested parties over a multicast connection. Thus, redundant and extraneous transmissions are filtered out, resulting in fewer packets sent over the network, which reduces network congestion.

Difficulties exist in creating a single protocol that meets the needs of all simulation applications. Therefore, key components of some simulations may not be supported by DIS and therefore cannot be represented over the network. The High Level Architecture is a more flexible architecture because it lets programmers define their own set of entity attributes and interactions called the Federation Object Model (FOM).

DIS also has no provisions for time management. Each simulation runs in real time and sends its updates over the network. Since each simulation runs independently, synchronization problems can occur between simulations making it nearly impossible for each simulation to maintain a consistent view of the virtual environment. HLA does provide support for time management.

E. OTHER APPROACHES

The computer entertainment industry has developed other architectures to network virtual environments. Computer game companies are capable of hosting massive multiplayer games online through the use of DirectX or similar technologies. These multiplayer games are based on a client/server architecture where the data for the virtual world resides on the server. As a player moves into a new area of the virtual world, the client downloads the world data from the server. Interactions between players or

other entities are routed through the server to other players in the same area.

While this architecture works well for games, it is not suitable for military training networked simulations at this time. To ensure adequate network performance, there is a limit to the level of detail in the virtual world that can be downloaded to a client in a reasonable amount of time. While game players are happy with the level of detail provided in games, the level of detail is not adequate for military applications.

In HLA applications, each simulation maintains its own database of the virtual environment. Changes in the environment can be distributed among the simulations. Since each simulation maintains its own model of the virtual environment, the level of detail can be much greater providing for a higher fidelity virtual environment.

THIS PAGE INTENTIONALLY LEFT BLANK

III. HIGH LEVEL ARCHITECTURE

A. OVERVIEW

The High Level Architecture was developed to establish

> a common high-level simulation architecture to facilitate the interoperability of all types of models and simulations among themselves and with C4I systems. The HLA is designed to promote standardization in the M&S community and to facilitate the reuse of M&S components.[3]

An HLA application includes a federation composed of one to many federates. A federation is group of simulations that interact in the same virtual environment. A federate is a member simulation of a federation. Federates communicate with each other through the Run Time Infrastructure (RTI). Federates register their entities and subscribe to entities of interest with the RTI. The RTI controls the data transfer between federates. The RTI is responsible for ensuring data is sent from the publishing federates to the subscribing federates.

The HLA architecture consists of three components.[4]

- Federation Rules
  - o Ensure proper interaction of simulations in a federation.
  - o Describe the simulation and federate responsibilities.

[3] Ibid, p. 1-2.

[4] Ibid, p. 1-3.

- Interface Specification
    - Defines Run-Time Infrastructure services.
    - Identifies callback functions each federate must provide.
- Object Model Template (OMT)
    - Provides a common method for recording information.
    - Establishes the format of key models:
        - Federation Object Model (FOM)
        - Simulation Object Model (SOM)
        - Management Object Model (MOM)

B. FEDERATION RULES[5]

1. Federation Rules:

1. Federations shall have an HLA FOM, documented in accordance with the HLA Object Model Template OMT.

2. In a federation, all representation of objects in the FOM shall be in the federates, not in the RTI.

3. During a federation execution, all exchange of FOM data among federates shall occur via the RTI.

4. During a federation execution, federates shall interact with the RTI in accordance with the HLA interface specification.

5. During a federation execution, an attribute of an instance of an object shall be owned by only one federate at any given time.

[5] Ibid, pp. 1-3 and 1-4.

2. Federate Rules:

6. Federates shall have an HLA SOM, documented in accordance with the HLA Object Model Template (OMT).

7. Federates shall be able to update and/or reflect any attributes of objects in their SOM and send and/or receive SOM object interactions externally, as specified in their SOM.

8. Federates shall be able to transfer and/or accept ownership of an attribute dynamically during a federation execution, as specified in their SOM.

9. Federates shall be able to vary the conditions under which they provide updates of attributes of objects, as specified in their SOM.

10. Federates shall be able to manage local time in a way that will allow them to coordinate data exchange with other members of a federation.

C. INTERFACE SPECIFICATION

The interface specification determines how federates interact with the federation through the RTI. The specification consists of six management areas.

1. Federation Management

Figure 1. Federation Management. (From: ref. 2)

2. Declaration Management

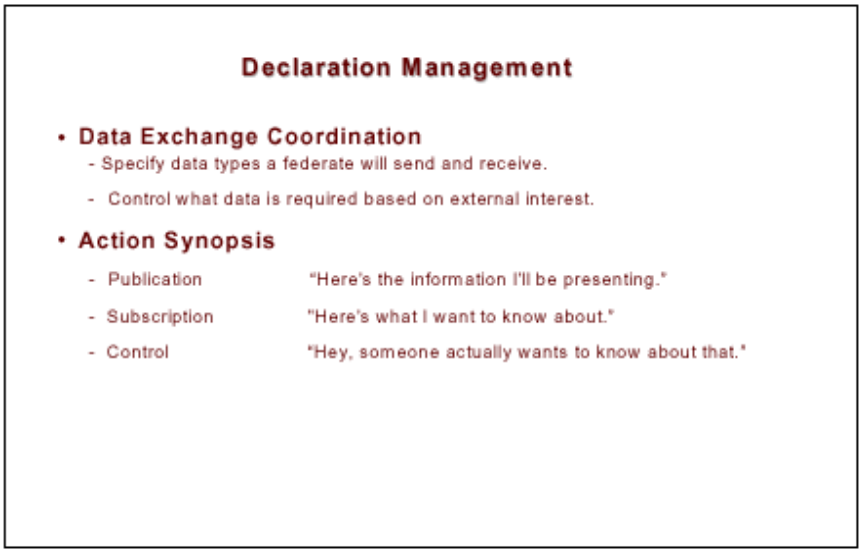

Figure 2. Declaration Management. (From: ref. 2)

3. Object Management

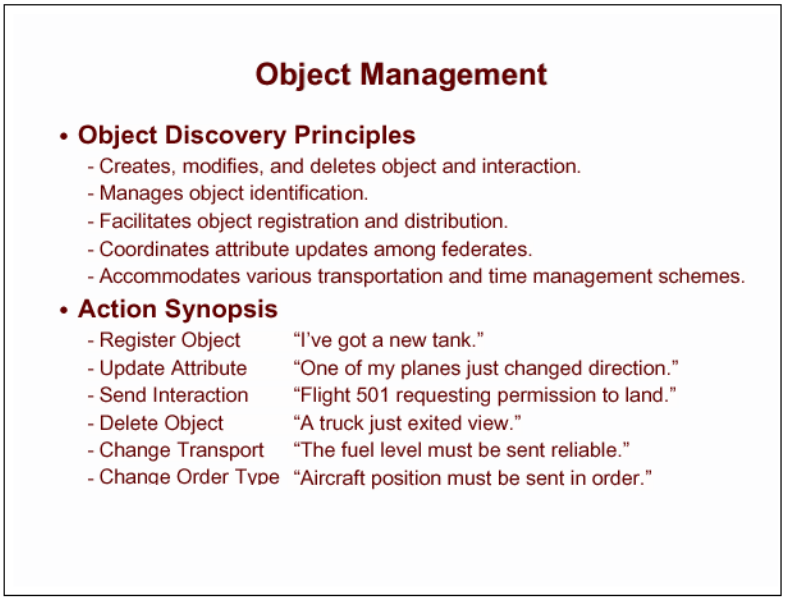

Figure 3. Object Management. (From: ref. 2)

4. Ownership Management

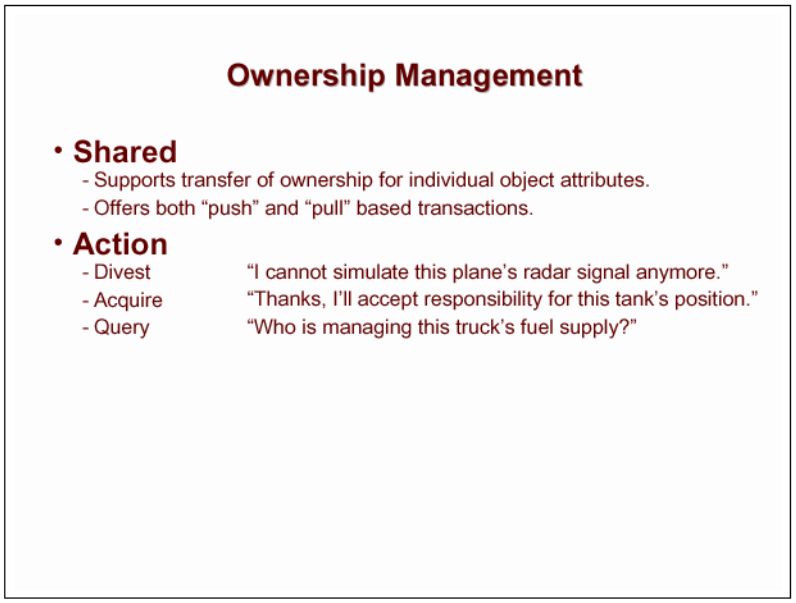

Figure 4. Ownership Management. (From: ref. 2)

5. Time Management

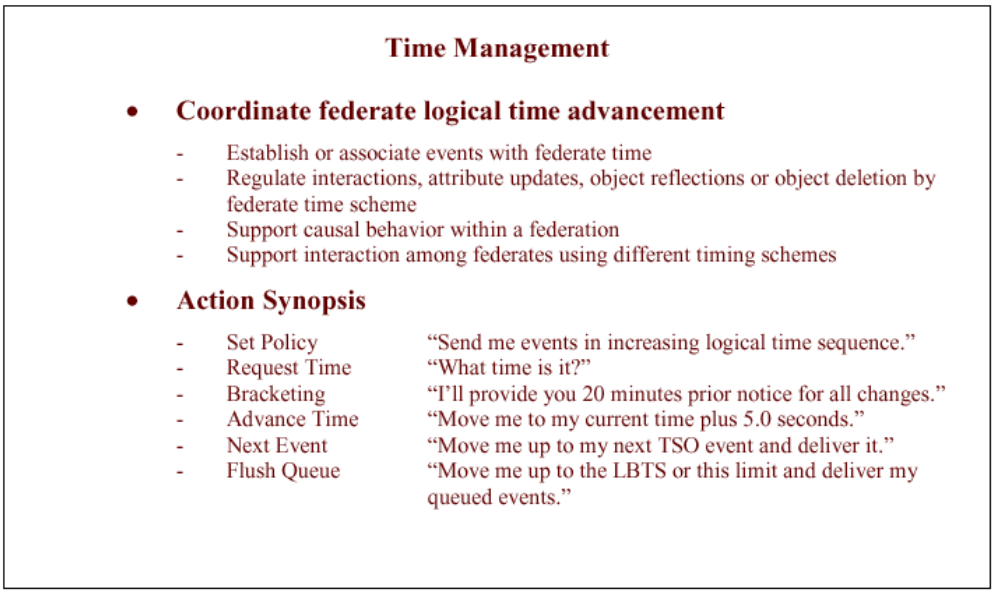

Figure 5. Time Management. (From: ref. 2)

6. Data Distribution Management

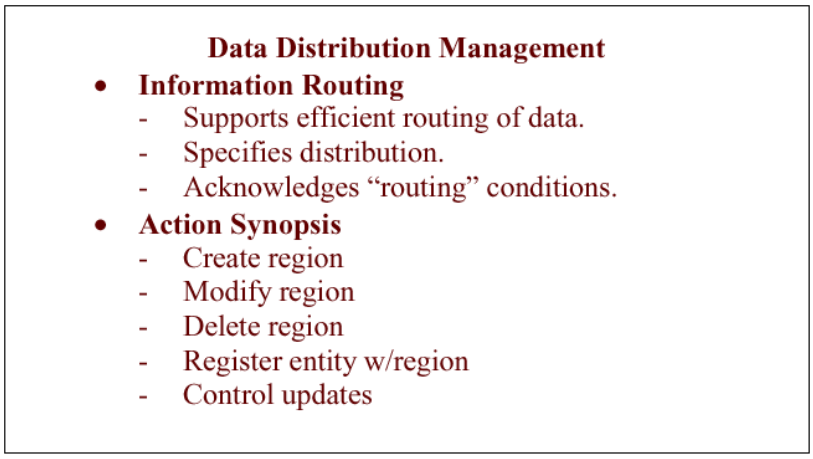

Figure 6. Data Distribution Management. (From: ref. 2)

D. OBJECT MODEL TEMPLATE (OMT)

The OMT establishes a common framework for object and interaction model documentation. The standard OMT provides a common method for describing HLA Object Models.

A FOM is an object model common to all federates in a federation. When a federation is created, the RTI reads the FOM in order to know what objects and interactions to expect. All federates in a federation must use the same FOM so that the RTI can coordinate and control the data transfer between federates. If a federate were able to use a different FOM, then that federate would not be able to recognize the objects and interactions that are passed back and forth.

The FOM is composed of two classes: objects and interactions. Objects are entities in the federate that have persistence. Examples of objects are tanks, aircraft, and ships. Attributes are used to describe an object. Examples of attributes are world position, orientation, and velocity. Interactions are non-persistent occurrences such as collisions, munition detonations, and weapons fire notifications. Interactions are made up of parameters. Examples of parameters are detonation location and detonation result.

An OMT for a specific FOM defines the objects and interactions for a FOM. For each object, the OMT defines that objects attributes. For each interaction, the OMT defines its parameters. Further, the OMT defines the data type of the parameters and attributes, cardinality (size of an array or sequence), units, and accuracy (maximum deviation from its intended value in the federate).

E. RUN TIME INFRASTRUCTURE

The RTI used for this thesis is the Defense Modeling and Simulation Office RTI 1.3-Next Generation Version 6. This RTI was chosen because it is the RTI used by JSAF.

The RTI Executive (RtiExec), the Federation Executive(FedExec), and the RTI libraty (libRTI) are the three components that make up the RTI.

1. RTI Executive

The RtiExec is a process that manages multiple Federation Executions in a network. Running the rtiexec.exe program starts the RtiExec. The RtiExec listens on an established multi-cast port for requests from federates to create and destroy federations. When a request for a federation creation is received, the RtiExec spawns a federation execution process to manage that federation. Requests from federates to join or resign a federation are directed to the appropriate FedExec by the RtiExec.

2. Federation Executive

The FedExec manages multiple federates in a federation. The FedExec processes join and resign requests from federates. The FedExec also controls and coordinates the data transfer between federates.

3. RTI Library

The RTI Library provides an interface to HLA services for the federates. The RTI Library contains two ambassador classes that enable communication between the federation and the federates: The RTIambassador class and the FederateAmbassador class. The RTI Library also contains

supporting classes and types that facilitate data transfer, see Appendix C of the HLA Programmer's Guide[6] for documentation of these classes and types.

a. RTIambassador Class

All requests from a federate to the RTI are made by making method calls to the RTIambassador class. Federates must declare an instance of an RTIambassador in order to communicate with the RTI. Appendix A of the HLA Programmer's Guide[7] contains descriptions of the methods of the RTIambassador class.

b. FederateAmbassador Class

The FederateAmbassador class is an abstract class in libRTI that must be implemented by each federate. The libRTI FederateAmbassador identifies callback functions that each federate must support. The RTI uses these callback functions to send data to the federates. Appendix B of the HLA Programmer's Guide[8] contains descriptions of the methods that must be supported by federate implementations of the FederateAmbassador class.

[6] Ibid. Appendix C.

[7] Ibid. Appendix A.

[8] Ibid. Appendix B.

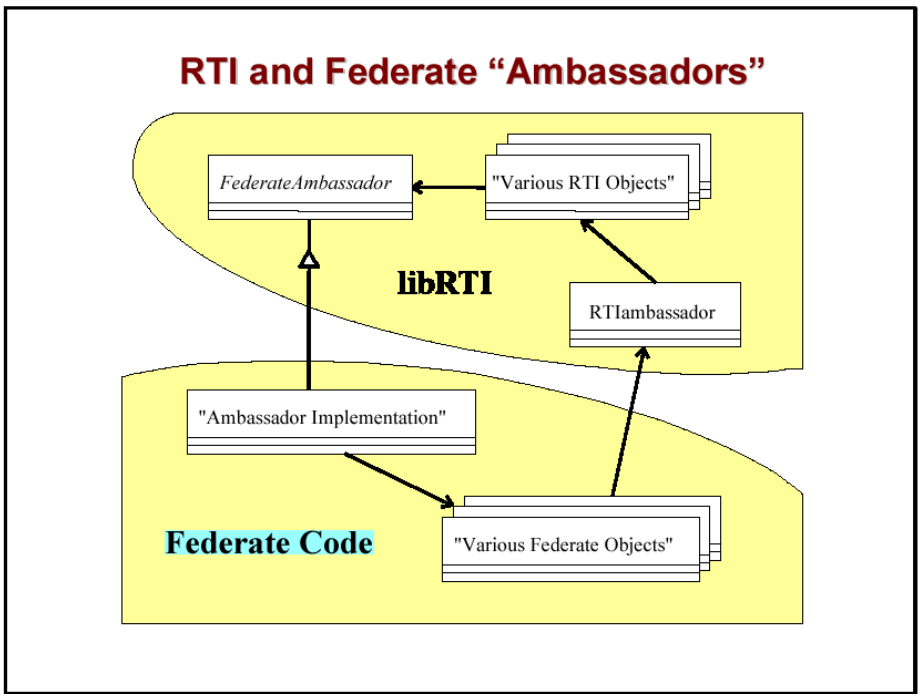

Figure 7. RTI and Federate Code Responsibilities. (From: ref. 2)

F. BASIC SEQUENCE OF EVENTS IN A FEDERATION

An HLA simulation follows a sequence of events from federation creation to destruction. A FedExec is created when a federate makes a create federation call to the RtiExec using an RTIambassador. During the process of creation the FedExec reads in the FOM for the federation, so that the federation will know what kind of objects and interactions can be expected during the simulation. After successful creation of a federation the federate makes a call to the FedExec to join the federation. The federate then publishes object attributes and interactions to the FedExec that the federate is capable of producing. The federate then creates and registers objects with the FedExec. The federate also subscribes to object and interactions types in the FOM that the federate is

interested in receiving from other federates in the federation. As other federates register their objects with the federation, the FedExec makes object discover calls to the FederateAmbassador of the federate. As the simulation progresses, the federate sends object attribute updates and interactions to the FedExec so that they can be distributed to other federates. The FedExec sends object attribute updates to the federate via the FederateAmbassador. During the simulation objects can be destroyed and therefore must be deleted from the federate. When the federate shuts down, it resigns from the FedExec. Finally, the last federate to leave the federation makes a call to destroy the federation.

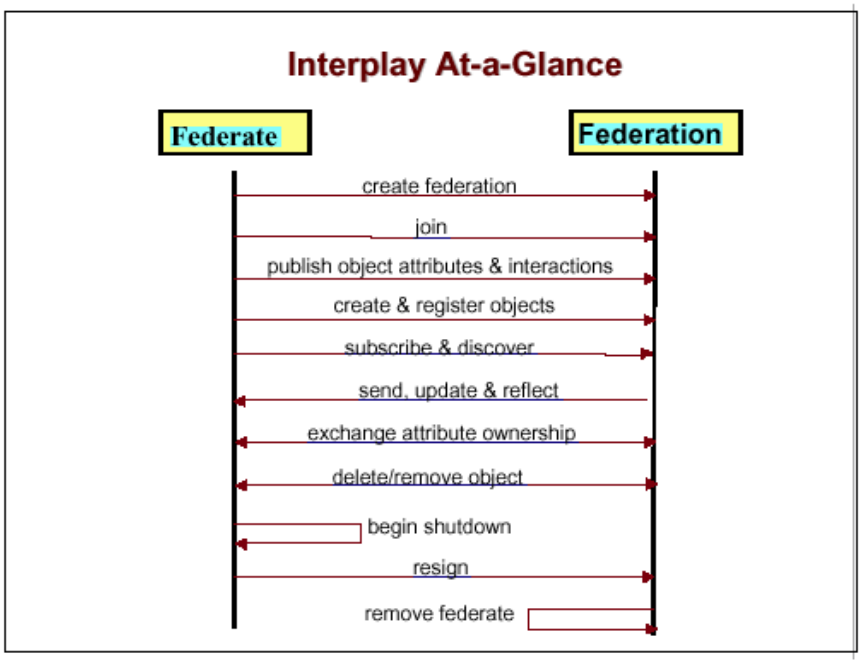

Figure 8. Federate and Federation Interplay. (From: ref. 2)

THIS PAGE INTENTIONALLY LEFT BLANK

IV. IMPLEMENTATION

The implementation of this thesis was completed in conjunction with Southwest Research Institute. This thesis was implemented with a C++ application and the DMSO RTI 1.3NG version 6. The application demonstrates an HLA implementation that supports the basic HLA services needed to run an HLA simulation. Services supported include: federation creation, join federation, resign federation, federation destruction, object and interaction publication, object and interaction subscription, object creation, object registration, sending and receiving object attribute updates, and sending and receiving interactions. These services will be covered in more detail below.

A. HIGH LEVEL ARCHITECTURE MODULE DESIGN

The HLA module consists of several classes. All of the HLA module classes begin with hm to help identify them as members of the HLA module. The hmDisplayController class controls the function calls to the rendering system. The hmHLAController class controls and coordinates services of the HLA. The hmFederateAmbassador class is the implementation of the RTI virtual class FederateAmbassador. The hmFederateAmbassador class receives communications from the RTI. The hmHLAObjectClass class will have one instance for each type of object that will interact with the RTI and will contain type wide attributes for that object type. The hmHLAObject class represents individual objects that interact with the RTI. The hmInteractionClass class handles interactions such as collisions and weapons fires. The hmHandleValuePair class is used to pair the RTI handle

for an attribute or parameter with its value. HLA objects use lists of hmHandleValuePairs to represent their attributes.

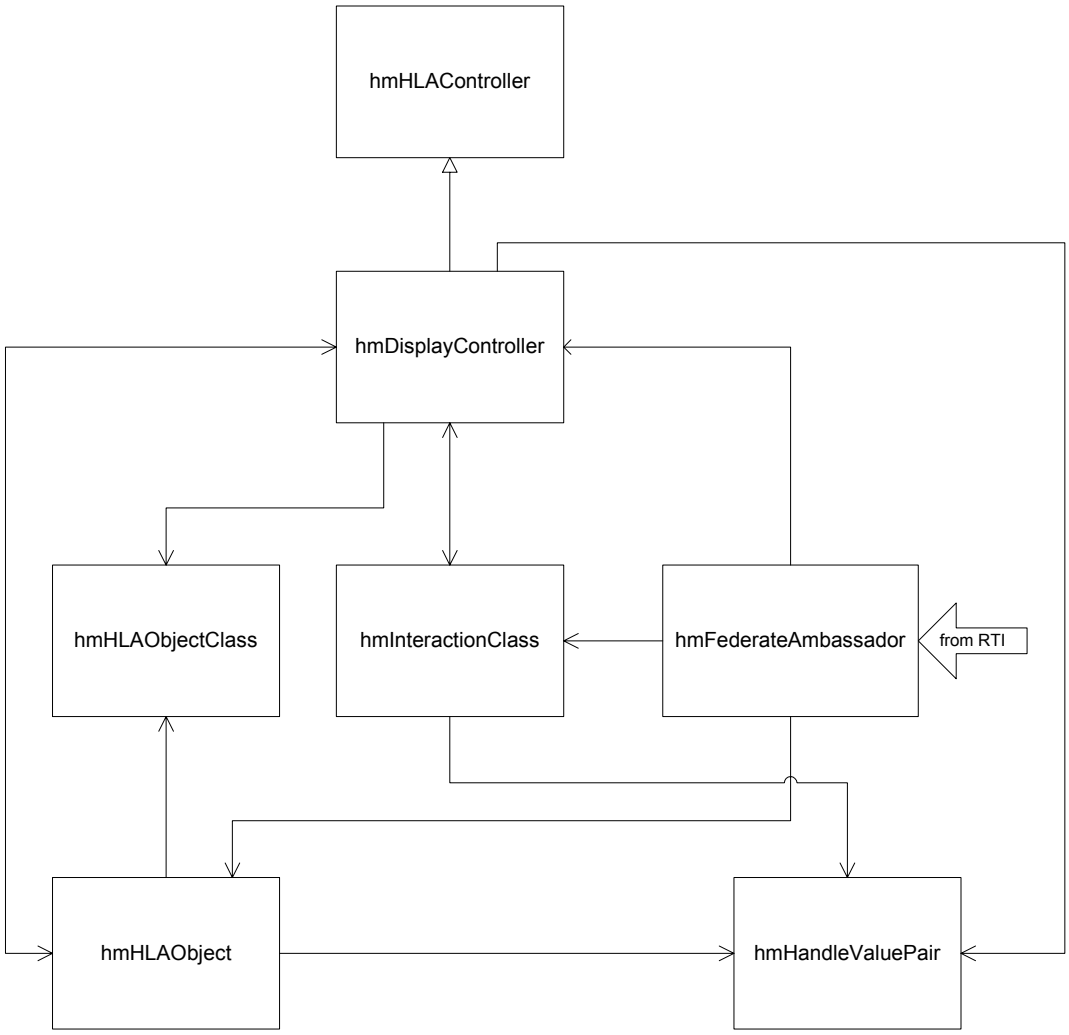

Figure 9. Class Relationship Diagram.

1. hmHLAController Class

The hmHLAController class coordinates HLA services. The class constructor creates a federation in the RTI if a federation by the same name has not already been created.

The constructor also joins the federate to the federation. The destructor resigns a federation and destroys a federation when no other federates are joined to a federation. The hmHLAController functions coordinate the transfer of data between the RTI federation and the federate.

2. hmDisplayController Class

The hmDisplayController class is inherited from the hmHLAController Class. The hmDisplayController class coordinates the transfer of data for the HLA module. The hmDisplayController class is the interface to a larger application. The hmDisplayController class makes all function calls to the rendering engine Application Programmer's Interface (API) for the HLA module, VEGA in this case. Since hmDisplayController is the only class in the HLA module that makes calls to the rendering engine, it is the only class that must be adjusted when switching to a different rendering engine.

Only one instance of an hmDisplayController should be declared in each federate in a federation. As it is written now, the hmDisplayController constructor initializes Vega, which only needs to be done once. Since hmDisplayController is inherited from hmHLAController, when an instance of hmDisplayController is declared the constructor for hmHLAController is also called. The hmHLAController constructor goes through the steps required to create and join a federation, which again only needs to occur once per federate.

Since hmDisplayController inherits from hmHLAController, no instance of an hmHLAController is

declared in the application. The hmDisplayController has access to all the HLA service functions of the hmHLAController, so all HLA service calls in the application call the hmDisplayController instance. The hmDisplayController class has overloaded functions for the functions in hmHLAController that require communication with the rendering engine, such as receiving object attribute updates from the RTI.

3. hmFederateAmbassador

The hmFederateAmbassador class is the HLA module implementation of the pure virtual class FederateAmbassador found in libRTI. The hmFederateAmbassador is the means by which the RTI federation communicates with the federate. This application only supports the object management functions of the FederateAmbassador Class. The hmFederateAmbassador keeps a pointer to the hmHLAController for the federate the hmFederateAmbassador is associated with. The hmHLAController pointer actually points to an hmDisplayController instance because no instance of an hmHLAController exists in this application. Since the hmDisplayController class inherits from the hmHLAController class, polymorphism allows an hmHLAController pointer to point to an hmDisplayController instance.

4. hmHLAObjectClass

The hmHLAObjectClass class is responsible for managing the different types of objects in the federate. The hmHLAObjectClass handles the object type wide services such as publication and subscription. The hmHLAObjectClass will have one instance for each type of object in the federate. The class maintains a static list of all of its instances.

The hmHLAObjectClass class maintains lists of published attributes, published object types and subscribed object types. The hmHLAObjectClass class keeps a member variable to hold the RTI ObjectClassHandle so that each instance knows the handle the FedExec uses to identify it. The hmHLAObjectClass class also maintains a pointer to the federate's RTIambassador.

5. hmHLAObject

The hmHLAObject class is responsible for managing the individual HLA objects in the federate. This class handles services such as sending and receiving attribute updates. The hmHLAObject class maintains a static list of all instances of the class.

Each hmHLAObject maintains several member variables in order to carry out its functions. The p_Handle variable is an RTI ObjectHandle, which is what the object is known as in the FedExec. Each hmHLAObject also keeps a handle to its object class, so that it knows what type of object it is. Another important member variable is a pointer to a visual object. The p_VisualObjPtr is a pointer to the object in the rendering system that represents this HLA object. This variable is important because when an update is received from the FedExec, the HLA object knows which rendering system object must be updated. This variable is stored as a void pointer to keep it general so that other rendering engines can be used without having to change the hmHLAObject class. Each object also keeps a pointer to the RTIambassador for the federate.

6. hmHLAInteractionClass

The hmHLAInteractionClass class is responsible for managing interaction services. This class processes the publishing, subscribing, sending and receiving for the different types of interactions supported by the federate. The hmHLAInteractionClass class maintains a list of its instances. The hmHLAInteractionClass maintains lists of published and subscribed interaction classes. This class also has a pointer to the federate's RTIambassador. Each instance keeps an RTI handle for its interaction class.

7. hmHandleValuePair

The hmHandleValuePair class matches the RTI handle for an object or interaction to the value held by that object or interaction. Lists of hmHandleValuePairs are used to process the sending and receiving of object attribute updates and interactions. Lists of hmHandleValuePairs are used so that the hmHLAObject and hmInteractionClass classes can process different types of objects and interactions. This also means that different FOMs can be easily supported. As long as the hmHandleValuePair class can handle the data types of the FOM, the HLA module can process the objects and interactions of any FOM. Current data types supported by the hmHandleValuePair class include string, integer, float, double, a C++ struct consisting of three floats, and a C++ struct consisting of three doubles. Another advantage is that only one class for objects and interactions needs to be written. The hmHLAObject and hmInteractionClass classes can process different types of objects and interactions.

B. HIGH LEVEL ARCHITECTURE SERVICES

The basic services of creating, destroying, joining, and resigning a federation executive are handled in the hmHLAController class by making appropriate calls to the RTI using an RTIambassador.

The processes for handling object and interaction services are detailed below.

1. Publishing Object Attributes

When a federate joins a federation execution, it must inform the FedExec of the types of objects the federate will be producing. It must also specify the attributes of those objects for which the federate will be sending updates.

To publish object attributes, first a call is made to the hmDisplayController function PublishObject (function inherited from hmHLAController) that takes string and vector of strings as its parameters. The first parameter is the name of the object class taken from the FOM. The second parameter is a list of the names of the attributes being published. The names of the attributes are also taken from the FOM.

The PublishObject function first searches the instance list of the hmHLAObjectClass for an instance with the same class name. If an instance is not found, then a new instance is created. Next, a call is made to the hmHLAObjectClass’s Publish function.

The Publish function takes the input parameter handle list and converts it to an AttributeHandleSet from libRTI. The function then makes a call to the RTIambassador’s

publishObjectClass function with the class handle and the AttributeHandleSet as a parameter and publishes the attributes in the FedExec. Next, the handle list and the AttributeHandleSet are added to the list of published attributes. Lastly, the object class is added to the list of published object classes.

The process for subscribing to object types is very similar to the publishing process and follows the same logical flow.

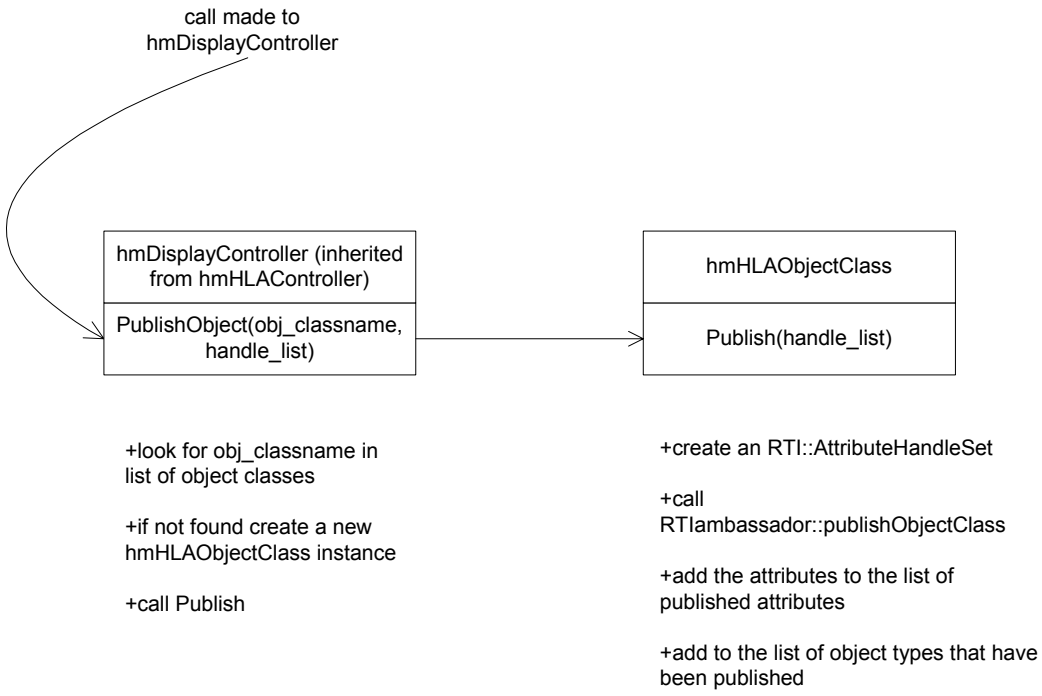

Figure 10. Publish Object Attributes.

2. Creating a Local Object

When a simulation creates an object to be displayed in the simulation and wants that object to be shared with the federation, an HLA object needs to be created and registered with the FedExec.

Before an hmHLAObject local to the federate can be created, the attributes it will be sharing must be published with the FedExec. To create a local hmHLAObject, a call is made to the hmDisplayController function CreateLocalObject (inherited from hmHLAController). The CreateLocalObject function takes two input parameters: a string and a void pointer. The first parameter is the object class name taken from the FOM. The second parameter is a pointer to the rendering system object cast to a void pointer. The CreateLocalObject function declares a new hmHLAObject and passes the class name string, a pointer to the RTIambassador, and a pointer to the visual object to the hmHLAObject constructor. When the CreateLocalObject function completes, it returns a handle to the newly created object.

The hmHLAObject constructor called from CreateLocalObject initializes the object with the input parameters. The constructor then registers the new object with the FedExec by calling the RTIambassador's registerObjectInstance function. Registering the object informs the FedExec of the object, so that the FedExec can process the object's updates.

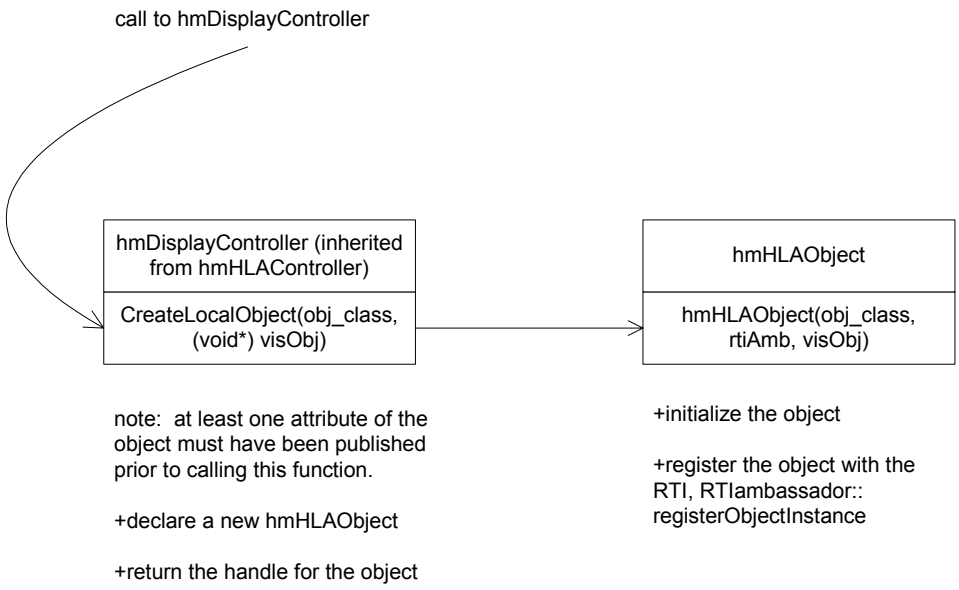

Figure 11. Create a Local Object.

3. Create a Remote Object

When the FedExec discovers a new instance of an object class that the federate has subscribed to, the FedExec calls the federate's FederateAmbassador function discoverObjectInstance. The discoverObjectInstance function takes three parameters: a handle for the object, a handle for the object's class, and a character string representing a FedExec designated name for the object.

First, the hmFederateAmbassador implementation of discoverObjectInstance checks the object class handle against the list of subscribed object classes to ensure that the object is of a type that the federate is interested in. If the object class is not in the subscribed class list, then an error message is displayed and the function terminates. If the object class is in the list of subscribed object classes, then the

CreateRemoteObject function of hmDisplayController (inherited from hmHLAController) is called.

The CreateRemoteObject function declares a new hmHLAObject and passes the object handle, object class handle, and a pointer to the RTIambassador to the constructor. The CreateRemoteObject then creates a new rendering system object that matches the object class, so the new object can be displayed in the simulation. The function then calls the SetVisualObj function of the hmHLAObject class with a void pointer to the new visual object as a parameter. The SetVisualObj function sets the visual object for the hmHLAObject instance.

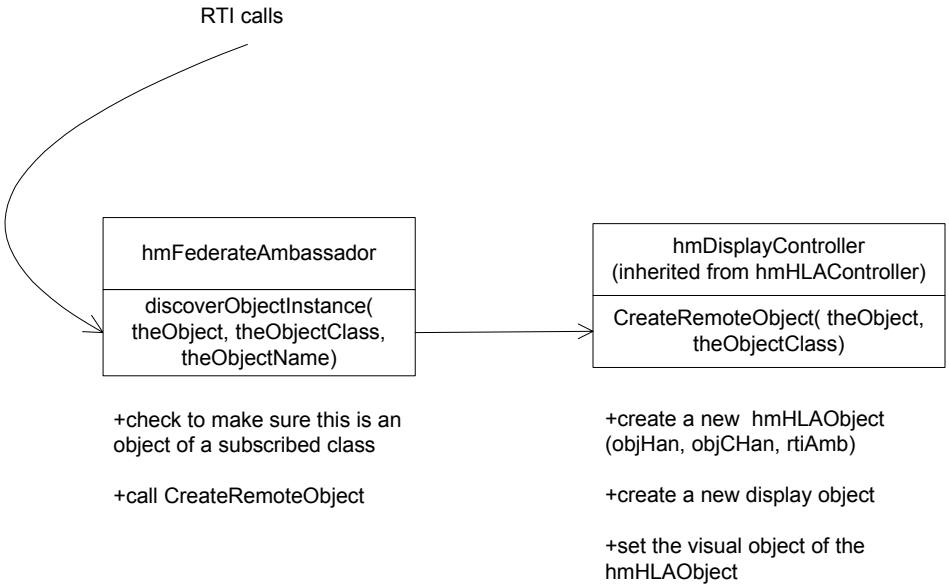

Figure 12. Create a Remote Object.

4. Send a Local Object Attribute Update

When an application decides to send an update to an object's attributes to the FedExec, the application builds a list of hmHandleValuePairs for the attributes to be updated. The hmHandleValuePairs contain the FOM attribute

name and the new value for that attribute. The application then calls the SendObject function of the hmDisplayController class (inherited from hmHLAController). The SendObject function takes two parameters: the hmHLAObject's handle and the list of hmHandleValuePairs. The SendObject function finds the hmHLAObject in the list of hmHLAObject instances and then calls that hmHLAObject instance's Send function.

The hmHLAObject function Send has just one parameter: the list of hmHandleValuePairs. First, the function converts the list of hmHandleValuePairs to an RTI AttributeHandleValuePairSet. The function then calls the RTIambassador function updateAttributeValues to send the updates to the FedExec. The updateAttributeValues function takes three parameters: the object handle, the AttributeHandleValuePairSet, and a character string tag.

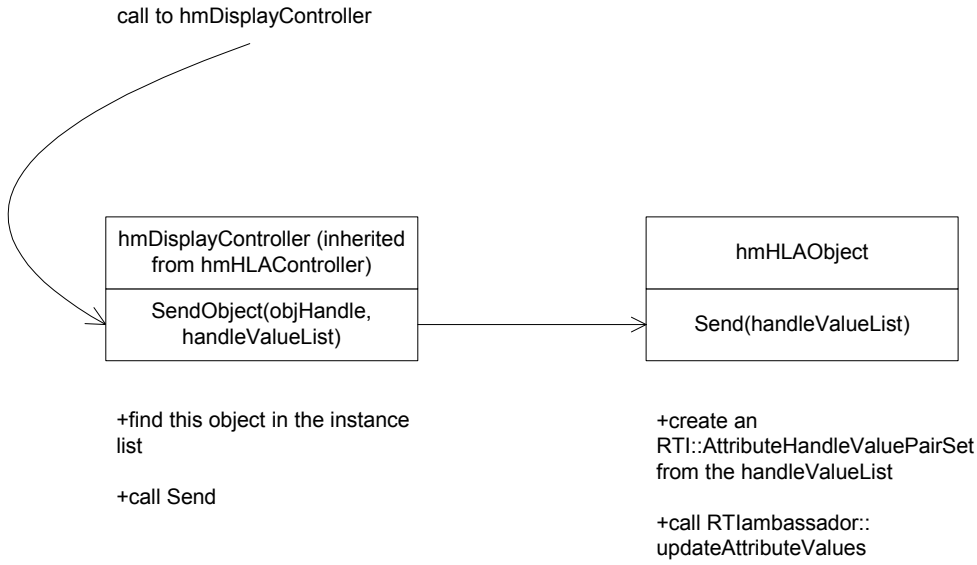

Figure 13. Send a Local Object Update.

5. Receive a Remote Object Attribute Update

When the FedExec receives an attribute update for object type that a federate has subscribed to, the FedExec makes a call to the federate's FederateAmbassador to reflect the attribute updates. The function called is reflectAttributeValues, which takes three parameters in this implementation. Those parameters are the handle to the object being updated, the AttributeHandleValuePairSet for the attributes being updated, and a character string tag.

The hmFederateAmbassador implementation of reflectAttributeValues first finds the object being updated in the list of hmHLAObject instances. The function then calls that instance's Receive function.

The hmHLAObject Receive function has two parameters: the AttributeHandleValuePairSet and a pointer to a hmHLAController (an hmDisplayController instance in this case). The hmHLAObject Receive function then converts the AttributeHandleValuePairSet into a list of hmHandleValuePairs. The Receive function then calls the hmDisplayController function ReceiveObjUpdate_cb (overloaded function from hmHLAController).

The ReceiveObjUpdate function takes two parameters: a pointer to the hmHLAObject being updated and the list of hmHandleValuePairs. The function determines the hmHLAObjectClass of the hmHLAObject, so that the function can properly update the rendering system object. The function then applies the appropriate updates.

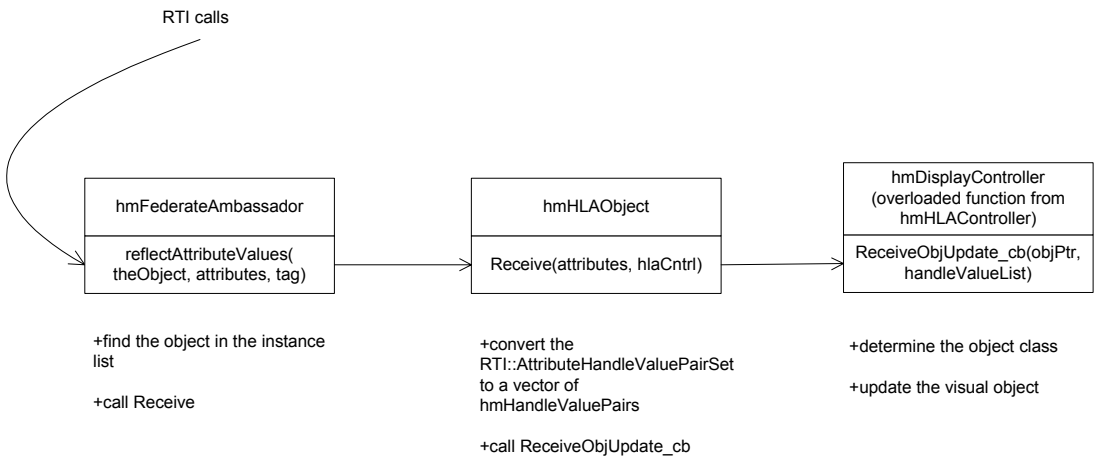

Figure 14. Receive a Remote Update.

6. Publish an Interaction

A federate must inform its FedExec what kinds of interactions the federate is capable of producing before it can start sending interactions to the FedExec. The federate informs the FedExec by publishing the types of interactions it can produce.

To publish a type of interaction, the application calls the PublishInteraction function of the hmDisplayController class (inherited from hmHLAController). This function has just one parameter: a string that is the interaction class name taken from the FOM. The PublishInteraction function first checks to see if a hmHLAInteractionClass instance exists with the class name input as a parameter to the function. If no such instance exists, the function declares a new hmHLAInteractionClass instance. The function then calls the hmHLAInteractionClass instance's Publish function

The hmHLAInteractionClass Publish function calls the RTIambassador function publishInteractionClass to publish

the interaction class with the FedExec. The function then adds the hmHLAInteractionClass instance to the list of published interaction classes.

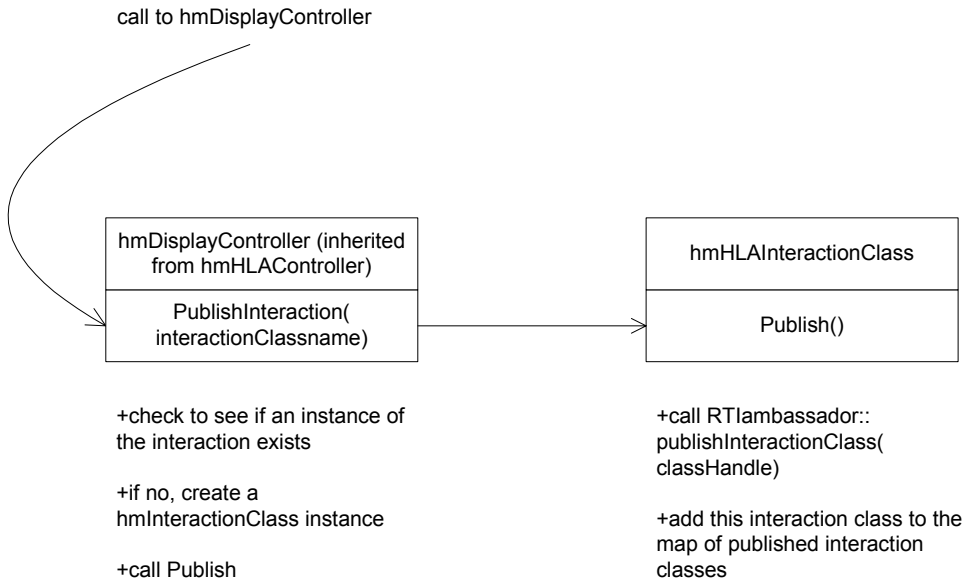

Figure 15. Publish an Interaction.

7. Send an Interaction

The framework exists in this implementation to send interactions, but currently no interactions are implemented in the test application.

When an interaction is generated by an application, the SendInteraction function of the hmDisplayController class is called (inherited from hmHLAController). The parameters of this function are a string representing the FOM name for the interaction class and a list of HandleValuePairs containing the parameters for the interaction. First, the function checks to see that this interaction class has been published. If the interaction

class has been published, then the function calls the hmHLAInteractionClass function Send.

The hmHLAInteractionClass function Send takes the hmHandleValuePair list as a parameter and then converts the list to an RTI ParameterHandleValuePairSet. The Send function then calls the RTIambassador function sendInteraction. The sendInteraction function has the following parameters: a handle to the interaction class, the ParameterHandleValuePairSet, and a character string tag.

The process for subscribing to interaction classes is very similar to publishing interaction classes.

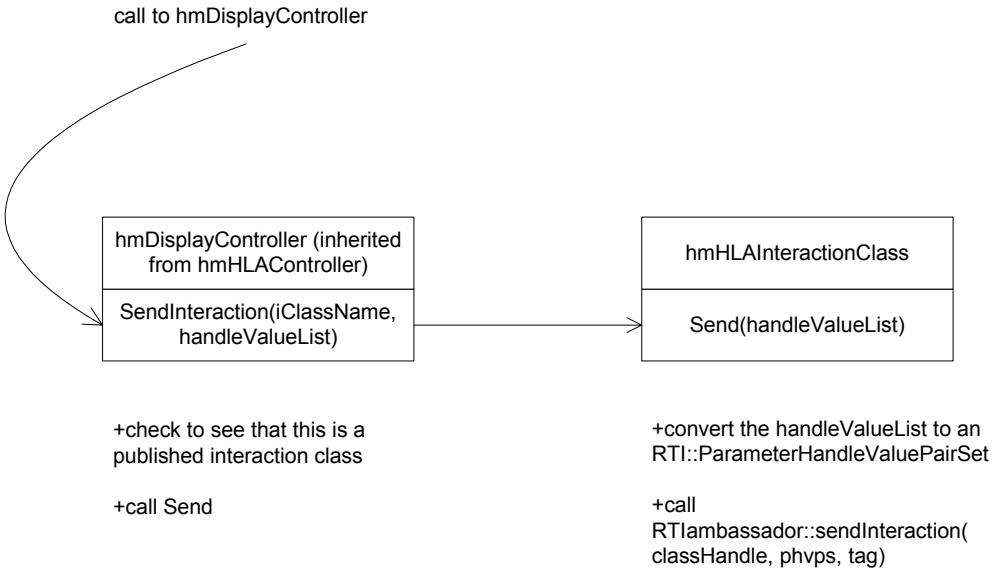

Figure 16. Send an Interaction.

8. Receive and Interaction

The framework exists in this implementation to receive interactions, but currently no interactions are implemented in the test application.

When the FedExec receives an interaction of a type that the federate has subscribed to, the FedExec makes a call to the federate's FederateAmbassador function receiveInteraction. The receiveInteraction function has the following parameters: the handle to the interaction class, an RTI ParameterHandleValuePairSet of the interaction's parameters, and a character string tag.

This application's hmFederateAmbassador implementation of receiveInteraction first finds the hmHLAInteractionClass instance from the hmHLAInteractionClass instance list that corresponds to the received interaction. The function then calls the hmHLAInteractionClass instance's Receive function.

The hmHLAInteractionClass Receive function has two parameters: the ParameterHandleValuePairSet and a pointer to an hmHLAController (an hmDisplayController instance in this case). The Receive function takes the input ParameterHandleValuePairSet and converts it to a list of hmHandleValuePairs. The function then calls the ReceiveInteraction_cb function of the hmDisplayController class (overloaded function of hmHLAController).

The ReceiveInteraction_cb function takes the following parameters: a pointer to the hmHLAInteractionClass instance and the list of hmHandleValuePairs. The ReceiveInteraction_cb function processes the interaction depending on what type of interaction is received.

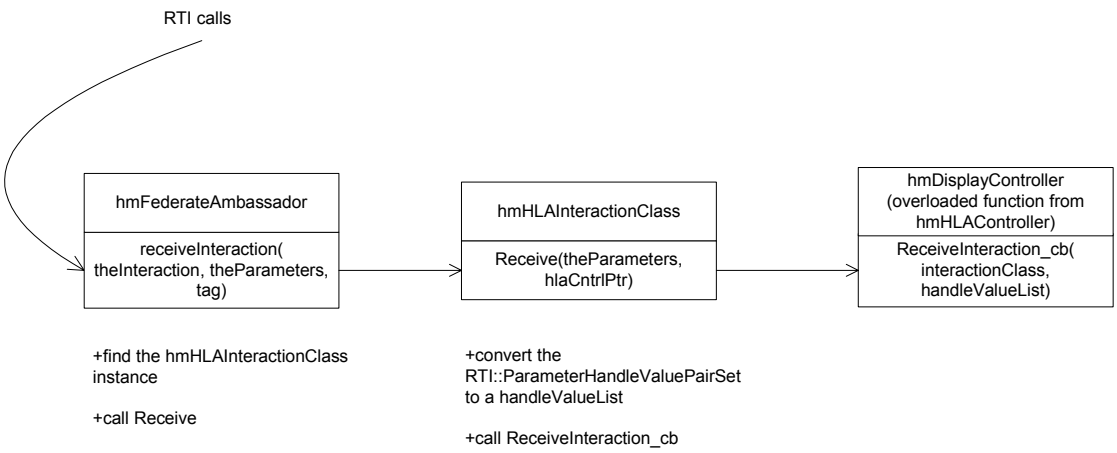

Figure 17. Receive an Interaction.

C. OBJECT MODEL

The Federation Object Model chosen was the Real-time Platform Reference Federation Object Model (RPR FOM) Version 1. The RPR FOM was developed by the Simulation Interoperability Standards Organization, Inc. (SISO). Details of this object model can be found in the *Guidance, Rationale, and Interoperability Modalities for the Real-time Platform Reference Federation Object Model* (GRIM RPR FOM).[9] This FOM was chosen for its wide usage and its compatibility with Joint Semi-Autonomous Forces (JSAF). The FedExec reads the RPR FOM from the rpr-1.0.fed file.

The RPR FOM was designed to provide Distributed Interactive Simulation (DIS) attribute and interaction functionality for an HLA object environment. The RPR FOM was designed to help transition DIS applications to HLA. The RPR FOM was also designed to provide a general framework to enhance interoperability.

[9] Reilly, Sean and Briggs, Keith. (1999). *Guidance, Rationale, and Interoperability Modalities for the Real-time Platform Reference Federation Object Model (RPR-FOM),* Version 1.0, SISO, inc.

Objects and interactions are maintained in a structured hierarchy in the RPR FOM. The RPR FOM object class structure is a four-tier hierarchy. Objects inherit the attributes of the objects in higher tiers of which the object is a child. For example, an aircraft will have attributes unique to an aircraft as well as the attributes of a platform, physical entity, and a base entity.

This thesis supports three object classes in the test application. The supported classes are Aircraft, AmphibiousVehicle, and GroundVehicle. All three classes inherit from the Platform class, which in term inherits from the PhysicalEntity class. The PhysicalEntity class inherits from the BaseEntity class. For the test application, two attributes were supported for these object classes: WorldLocation and Orientation. WorldLocation and Orientation are both attributes of BaseEntity, so the three object classes inherited these attributes. The WorldLocation attribute describes an object's location in the simulation by giving x, y, and z coordinates in meters. WorldLocation is represented as a C++ struct of three doubles. The Orientation attribute describes the object's orientation in space. The object's orientation is described by three angles: Psi or heading, Theta or pitch, and Phi or roll. The units for the three angles are in radians. The Orientation attribute is represented as a struct of three floats.

Interactions in the RPR FOM are structured in a three-tier hierarchy. Collision of the EntityInteraction family and MunitionDetonation and WeaponFire of the Warfare family

would be the most commonly used interactions. However, no interactions are fully supported in the test application.

D. COMPATIBILITY WITH JOINT SEMI-AUTONOMOUS FORCES

This thesis was designed to be compatible with JSAF. The main reason the RPR FOM was chosen as the FOM for this thesis is because JSAF supports it. Additionally, the RTI used in this thesis is the same as the one used by JSAF. However, this thesis did not test compatibility with JSAF in the test application.

E. CHANGING RENDERING PLATFORMS

The HLA module was designed so that only the hmDisplayController class needs to be changed when changing rendering platforms. In the test application, the hmDisplayController class is the only class that makes calls to the VEGA API and the hmDisplayController does not makes calls directly to the RTI.

Several hmDisplayController functions would need to be changed to support a new rendering platform. The hmDisplayController constructor would need to be changed to initialize the new rendering system and its variables. The two call back functions for receiving attribute updates and interactions would need to be changed to process the updates for the new rendering platform. The CreateDisplayObject function would need to be changed, so that the new object created is an object from the new rendering system. Lastly, the real time loop in the Run function would need to be changed so that local object updates are generated from the new rendering system.

F. INTEGRATING THE HIGH LEVEL ARCHITECTURE MODULE INTO AN EXISTING APPLICATION

The hmDisplayController class is the interface to integrate an existing standalone application into an HLA supported networked virtual environment. The Run function of the hmDisplayController currently contains the runtime loop for the test application. An existing application could adjust the Run function to execute the application's runtime loop and make appropriate calls to the application's classes.

Another option available would be to use the application existing runtime loop and make appropriate calls to the hmDisplayController class to communicate with the RTI. In the second option, the hmDisplayController class will need to be able to make calls to the rendering engine API in order to manipulate the rendering engine objects.

For a large application with many supporting classes, using the application's existing run time loop would be preferred. In this case, making adjustments to the hmDisplayController class would be simpler than adapting the hmDisplayController Run function and possibly making changes to multiple supporting classes.

THIS PAGE INTENTIONALLY LEFT BLANK

V. TESTING AND RESULTS

A. PROTOTYPE SYSTEM

The initial prototype was a simple application to establish a working High Level Architecture (HLA) application. The initial prototype supported only one object type and VEGA code was integrated throughout the application. No interaction support was included in the prototype. A simple Federation Object Model (FOM) was used in the prototype

For the prototype, a federate application was run on each of two machines with the Run Time Infrastructure (RTI) executive running on a third machine. Each federate had one entity that was shared over the network. Each federate used the same terrain model. The RTI software was loaded on all three machines, so that the RTI libraries would be available locally on each federate machine. The initial prototype successfully linked the two federates. Both entities could be seen on each federate application.

Both computers used to test the initial prototype had dual one GHz Intel Pentium III processors and a GeForce 3 graphics card. Each federate achieved a frame rate of approximately 30 frames per second when running the HLA application.

For a comparison with an application run in a standalone mode, the LynX active preview tool was used to preview the initialization file for the VEGA application. The preview tool showed an average frame rate of around 75 frames per second.

B. FINAL DESIGN

From the prototype, further work was done to add support for additional object types and to isolate the rendering engine specific code. Also, the framework for supporting interactions was added. Work was also completed to change FOMs to the Real-time Platform Reference Federation Object Model (RPR FOM). This further work lead to the development of the final implementation design for this thesis.

The final design of the HLA module was tested using a simple application. Again, two computers were used to run one federate each. However, the RTI executive for this test was in another building on campus on the same network. One federate had an aircraft object while the other federate had an amphibious vehicle object. A federation was successfully created and joined by the federates. Each federate published and registered their local objects and subscribed to the object types each was interested in. Each federate successfully discovered the others object and correctly displayed the correct object type within the simulation. Position and orientation information were passed between the federates once per frame. Each federate successfully updated their remote object’s position and orientation. At the termination of the simulation, each federate correctly resigned from the federation and the federation was destroyed.

To test the federation simulation, one federate application was run on a computer with dual 500 MHz Intel Pentium III processors and an Intense3D Wildcat 4000 graphics card with 16 MB of video RAM. The other federate

was run on a laptop computer with a one GHz Intel Pentium III processor and an NVIDIA GeForce2 Go graphics card with 32MB video RAM. Frame rates averaged around 20 frames per second on both machines.

For a comparison with an application run in a standalone mode, the LynX active preview tool was used to preview the initialization file for the VEGA application. The preview tool showed an average frame rate of around 80 frames per second.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSION

A. GENERAL DISCUSSION

The intent of this thesis was to create a High Level Architecture (HLA) Module that would allow other programmers to quickly develop an HLA compliant simulation. To do this, the HLA module had to be as general as possible to allow programmers the flexibility to develop applications in whatever programming environment most suited their needs. This requirement meant that the core of the module had to be independent of the system used to render the simulation. Additionally, support for multiple object models was desirable, so attribute and interaction handling had to be generalized within the HLA module.

The HmDisplayController class is interface to rendering system and FOM. The hmDisplayController class makes necessary calls to the rendering system that relate to HLA services. The hmDisplayController class coordinates data flow in the HLA module; however, it makes no direct calls to the RTI. Handling of FOM data types are generalized within the HLA module by using lists of hmHandleValuePairs for processing.

B. CONTRIBUTIONS

The contribution made by this thesis is a framework on which to build HLA compliant networked virtual environments. This thesis provides support for the basic services required for any HLA application and a structure on which to add support for more HLA services, objects and interactions. Federation HLA Services supported include federation creation, join, resign, and destruction. Object

services include publishing, subscribing, creating, registering, discovering, and updating. Interaction services include publishing, subscribing, sending, and receiving.

C. FUTURE WORK

Several areas are available for further study in relation to this thesis. These areas include adding additional HLA services, support for more object types and attributes and interactions, and increasing application performance.

1. Additional High Level Architecture Services

For this thesis a minimum number of HLA services was provided. More service areas exist that could be supported. Time Management, Ownership Management, and Data Distribution Management were not supported at all in this implementation. Federation management services such as federate synchronization and saving and restoring federates could also be supported.

a. Time Management

Time Management is an important service provided for in HLA. Time Management allows federates to remain consistent with each ensuring all federates maintain the same world picture. With Time Management, time is advanced in a coordinated fashion. Object updates and interactions will have a timestamp in their packets; so that the receiving federates know exactly when an event has occurred. A time regulating federate is responsible for the progression of time in a constrained federate. By default, HLA applications are not time regulating or time constraining.

b. Ownership Management

A federate owns an object when that federate is able to send attribute updates for that object. Only one federate can own an object at any one time. Ownership Management provides methods for transferring ownership between federates.

c. Data Distribution Management

In a large simulation with many entities a federate may not care about updates for an entity a large distance away. Processing updates for an entity outside of a federate's sensor range would be extraneous and inefficient. Data Distribution Management allows for the creation of regions. With Data Distribution Management, a federate will only receive attribute updates and interactions that occur within the federate's region.

2. Additional Objects and Interactions

Only a minimal number of object types and no interactions were supported for this implementation. Of the object types supported only object position and orientation were supported. The Real-time Platform Reference Federation Object Model (RPR FOM) has defined many more object types, attributes and interactions. Additional object types that could be supported include sea vehicles, space vehicles, and multi-environment vehicles. Attributes such as both linear and angular velocity and acceleration would need to be supported for a federation using dead reckoning for example. Interactions such as collisions, weapon fires, and munition detonations should be supported at a minimum in most simulations.

3. Improved Network Performance

Currently, the test application for this thesis sends and receives object attribute updates at frame boundaries. This causes an excess amount of network traffic and is not scalable past more than just a few entities. A more intelligent method for sending updates is needed. Updates should only be sent when there is a change in attributes of an object. A dead reckoning algorithm should be implemented so that federates can continue to move objects along a projected path until a new update is received. A heart beat update packet could be sent every five seconds for objects with no changes in that time span so that federates new that the object still exists in the simulation as is done in Distributed Interaction Simulation applications.

LIST OF REFERENCES

[1] Alluisi, E. A. (Jun. 1991). The Development of Technology for Collective Training: SIMNET, a Case History. Human Factors, Training Theory, Methods, and Technology, Volume: 33, Issue: 3, 343-362.

[2] Ceranowicz, A. (no date). STOW, the Quest for a Joint Synthetic Battlespace. In Proctor, Michael D (Ed.). *Web-based Technical Reference on Simulation Interoperability* (Ch. 8). (online). Available: <http://www.engr.ucf.edu/people/proctor/Interoperability%20Text/Text%20Outline.htm> (29 Aug. 02).

[3] Cosby, L. N. (1995). SIMNET: an Insider's Perspective. In Clarke, T. L. (Ed.). SPIE Proceedings Vol. CR58, Distributed Interactive Simulation Systems for Simulation and Training in the Aerospace Environment (pp. 59-72).

[4] Davis, P.K. (Aug. 1995). Distributed Interactive Simulation in the Evolution of DOD Warfare Modeling and Simulation. Proceedings of the IEEE , Volume: 83 Issue: 8, 1138 –1155.

[5] Department of Defense, Defense Modeling and Simulation Office. (no date). *High Level Architecture RTI 1.3-Next Generation Programmer's Guide*, Version 5.

[6] Department of Defense. (1998). *High Level Architecture Interface Specification Version 1.3*, Draft 11.

[7] Department of Defense. (1998). *High Level Architecture Object Model Template Specification Version 1.3*.

[8] Department of Defense. (1998). *High Level Architecture Rules Version 1.3.*

[9] Loper, M. L. (1995). Introduction to Distributed Interactive Simulation. In Clarke, T. L. (Ed.). SPIE Proceedings Vol. CR58, Distributed Interactive Simulation Systems for Simulation and Training in the Aerospace Environment (pp. 3-16).

[10] Miller, D.C. & Thorpe J.A. (Aug 1995). SIMNET: the Advent of Simulator Networking. Proceedings of the IEEE, Volume: 83 Issue: 8, (pp. 1114 –1123)

[11] Ping, Ivan C. K. (2000). HLA Performance Measurement, Computer Science Department, Naval Postgraduate School, Mar 2000.

[12] Reilly, Sean and Briggs, Keith. (1999). *Guidance, Rationale, and Interoperability Modalities for the Real-time Platform Reference Federation Object Model (RPR-FOM),* Version 1.0, SISO, inc.

[13] Sandeep Singhal & Michael Zyda. 1999. Networked Virtual Environments - Design and Implementation, Reading, Massachusetts: Addison-Wesley.

APPENDIX A. C++ SOURCE CODE

The source code for this implementation will soon be available at http://libgf.sourceforge.net.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Marine Corps Representative
   Naval Postgraduate School
   Monterey, California

4. Director, Training and Education, MCCDC, Code C46
   Quantico, Virginia

5. Director, Marine Corps Research Center, MCCDC, Code C40RC
   Quantico, Virginia

6. Marine Corps Tactical Systems Support Activity (Attn: Operations Officer)
   Camp Pendleton, California